From System Comprehension to Program Comprehension

Christos Tjortjis, Nicolas Gold, Paul Layzell
Department of Computation, UMIST
Email: {christos, Nicolas.Gold
pjl}@co.umist.ac.uk

Keith Bennett
Department of Computer Science
University of Durham
Email: keith.bennett@durham.ac.uk

Abstract

Program and system comprehension are vital parts of the software maintenance process. We discuss the need for both perspectives and describe two methods that may be integrated to provide a smooth transition in understanding from the system level to the program level.

Results from a qualitative survey of expert industrial software maintainers, their information needs and requirements when comprehending software are initially presented. We then review existing software tools which facilitate system level and program comprehension.

Two successful methods from the fields of data mining and concept assignment are discussed, each addressing some of these requirements. We also describe how these methods can be coupled to produce a broader software comprehension method which partly satisfies all the requirements. Future directions including the closer integration of the techniques are also identified.

1. Introduction

Software maintenance accounts for the largest cost in the software lifecycle [22]. Within the process of software maintenance, program and system comprehension play a crucial and costly role [19]. Maintainers must understand not only the localised part of a program that they need to change, but also the context within which the change takes place – system understanding. Many support methods and tools in the field of program comprehension (the term is often applied to both program and system level comprehension) are focussed at one or the other. In this paper, we show how such methods may be coupled together to produce a more complete support environment for the software maintainer. This allows for switching between system and program views and partly satisfies all the requirements of industrial scale software comprehension.

The remaining of the paper is organised as follows: Section 2 presents the requirements of industrial software maintainers identified by a survey conducted in the U.K. Section 3 reviews existing software comprehension tools. Sections 4 and 5 present two methods for system and program level comprehension respectively. Section 6 discusses the extent to which these methods meet these requirements. Section 7 proposes ways for combining the methods so as to satisfy the complete set of requirements. Section 8 presents directions for further work.

2. Software maintenance requirements

Domain knowledge and expertise are crucial for software maintenance, the type of required knowledge changing over the lifetime of software. However it is recognised that there are no explicit guidelines given a program understanding task, nor are there good criteria to decide how to represent knowledge derived by and used for it [1]. A fundamental research challenge therefore was to understand the key industrial needs, objectives and assumptions in the program comprehension process and to provide the most appropriate support for the task at hand the time it is needed.

To determine the needs of software maintainers, understand their broad strategies, particularly the initial steps in program comprehension, and thereby provide better tool support, a qualitative survey of expert software maintainers was undertaken [26]. The survey confirmed that there is no high-quality substitute for experience when it comes to understanding a system, as existing methods and tools are not effective enough and documentation tends to be unreliable.

The main Software Maintenance practices and requirements identified by this survey were the following:

1. High level overviews, abstractions, localised system diagrams, module interrelationships and also means to estimate the impact of changes are required to be derived in an automated manner in order to accelerate and enhance program comprehension.
2. It was reported that program mental models, i.e. high level abstractions of subsystems with related functionality and interrelationships, are implicit in maintainers' work, but are hardly ever recorded for future use. The need for visualising, recording and cross-referencing these models in order to share experiences, improve communication and resolve misunderstandings was clearly identified.

3. Identification of a starting point for subsequent tracing through programs significantly accelerates the comprehension process. This normally occurs through consultation with experts and by use of maintainer's own experience but alternative means are essential.
4. Information exchange among team members is sparse, informal and is hardly ever recorded. There is a clear requirement for a means to provide standardised, reliable and communicable information regarding a system as an equivalent to knowledge available only to developers or experienced maintainers.
5. Maintenance is mainly documented in source code comments, except from extensive changes which are also reflected on user manuals. The implication is that comments in mature systems get accumulated over time and tend to reflect subsequent changes rather than the original implementation ideas. Capturing knowledge regarding past modifications by extracting information from comments and relating this to known functionality of code emerges to be of great importance.
6. The types of maintenance influence the approach taken. Corrections involve attempting first to locate the point where the fix needs to be applied. Enhancements require a 'detail-first' strategy, where a high-level understanding of the system's functionality and modules interrelationships is pursued before the change is made. Preventative maintenance was deemed rarely to occur and was considered to be an integral part of software development. The above highlight that maintainers are often required to switch between System Level and Program Comprehension.
7. Partial comprehension is pursued and achieved in most cases, which has to be balanced against the risk of failure in completing a maintenance task. It was reported that the time available for comprehension was limited because of commercial pressures and deadlines.

It was generally agreed that the most useful pieces of information to facilitate code comprehension are:

a. An easy to navigate, multi-layered subsystem abstraction and modules interrelationships providing an overview of the system and possible impact of changes.
b. Knowledge derived from past maintenance which can mainly be retrieved from comments.

3. Comprehension support

There are many types of tools available to help with software comprehension, emphasising different aspects of systems and modules, and usually creating new representations for them [10]. Biggerstaff et al. differentiate between naïve and intelligent agents (tools) for providing such representations [3]. Naïve agents generally perform deductive or algorithmic analysis of program properties or structure, e.g. program slicers [23] or dominance tree analysers [5]. Intelligent agents assign descriptions of computational intent to source code.

Biggerstaff et al. [3] claim that research on intelligent agents can be divided into 3 distinct approaches:

1) Highly domain specific, model driven, rule-based question answering systems that depend on a manually populated database describing the software system. This approach is typified by the Lassie system [8].
2) Plan driven, algorithmic program understanders or recognisers. Two examples of this type are the Programmer's Apprentice [20], and GRASPR [27].
3) Model driven, plausible reasoning understanders. Examples of this type include DM-TAO [3], [4], IRENE [17], and HB-CA [10], [12].

One exception to this categorisation is Hartman's work [14] that falls between approaches 2 and 3.

Systems using approaches 1 and 2 are good at completely deriving concepts within small-scale programs but cannot deal with large-scale programs due to overwhelming computational growth. Approach 3 systems can easily handle large-scale programs since their computational growth appears to be linear in the length of the program under analysis. They suffer from approximate and imprecise results [3].

Figure 1 is based on the summary of the program understanding landscape in [3] as extended in [10]. The original has been updated to include additional work on program understanding, with the number of each oval providing a key to the citations below. Citations have also been added to the original figure.

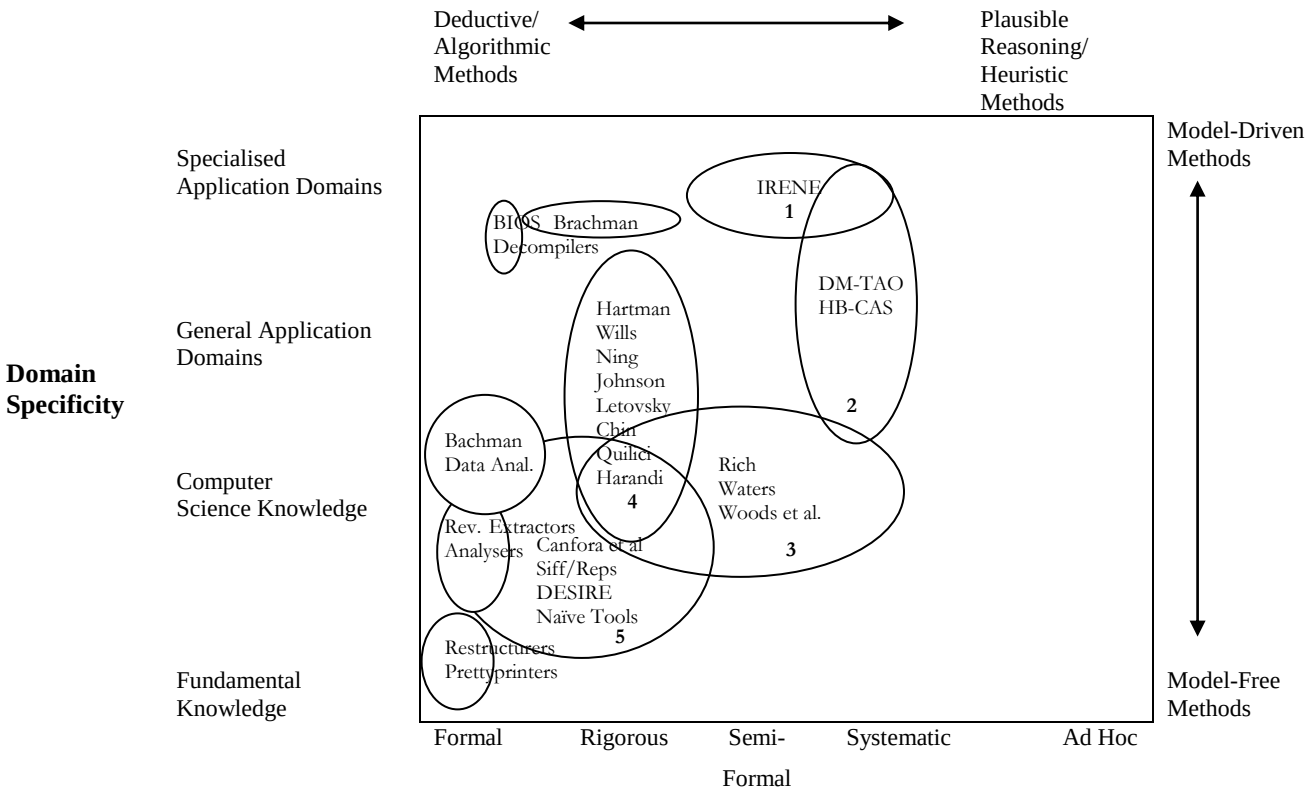
4. A method for system level comprehension

Data mining involves applying data analysis and discovery algorithms to data collections that produce a particular enumeration of patterns over the data [9]. Several techniques can give insight into vast amounts of data and extract useful, previously hidden knowledge.

Clustering is such a technique for partitioning a data set into mutually exclusive groups (clusters). Members of a cluster are similar to one another and dissimilar from members of other groups, according to some metric. Similarity is decided by measuring the distance of records with respect to all available variables [15].

Data Mining Code Clustering (DMCC) [25] is an approach, devised to address the need for automated methods providing a quick, rough grasp of a software system, to enable practitioners, who are not familiar with it, to commence maintenance with a level of confidence as if they had this familiarity.

DMCC primarily aims at providing a broad contextual picture of a system, rather than a detailed model [25]. This provides a roadmap by which maintainers can quickly navigate around the code, scoping the change



Key to citations

Oval	Author(s)	System	Citation(s)	Oval	Author(s)	System	Citation(s)
1	Karakostas	IRENE	[17]	4	Ning Kozaczynski	Concept Recogniser	[18]
2	Biggerstaff et al.	DM-TAO	[3], [4]	4	Johnson	PROUST	[16]
2	Gold	HB-CAS	[10], [11], [12]	4	Chin, Quilici	DECODE	[6]
3	Rich, Waters	Programmer's Apprentice	[20]	4	Harandi, Ning	PAT	[13]
3	Woods et al.	PU-CSP	[28]	5	Biggerstaff et al.	DESIRE	[2], [3], [4]
4	Hartman	UNPROG	[14]	5	Siff, Reps	FCA Tools	[21]
4	Wills	GRASPR	[27]	5	Canfora et al.	Various Methods	[6]

Figure 1: The program understanding landscape [10] after [3]

request and solution space. This enables more detailed analysis of targeted code to be undertaken.

DMCC portrays a program as a number of entities grouped in clusters representing subsystems, based on their similarity. Clusters indicate functions structure and interrelationships among them, in a way that the impact of changes can be predicted. A prototype tool for clustering C/C++ source code was developed, using functions as entities. Attributes include the use and types of variables / parameters and the types of returned values. Additional information about interrelationships among attributes is also used. Custom-made similarity metrics based on the association coefficient paradigm, were introduced and an agglomerative hierarchical clustering algorithm using the complete linkage method was employed.

The tool was evaluated using data extracted from C/C++ systems of various sizes. Experimental results indicate that a high-level system abstraction as a number of subsystems can be achieved by clustering program functions into groups. Interrelationships amongst components were identified in a similar manner. The accuracy of the results was evaluated by comparing the produced subsystem abstractions with experts' mental models. The abstractions were accurate, capturing the subsystems consistently with the mental model. Pair-wise values of precision and recall ranged between (50%, 40%) and (87%, 100%), i.e. highest precision achieved was 87% and highest recall 100% [24].

Grouping program components into subsystems reduces the perceived complexity thus facilitating maintenance. Corrective and adaptive maintenance is supported by the automatic derivation of a meaningful decomposition of source code into several subsystems, by identifying the interfaces between subsystems and determining the role each plays in performing a service [25]. This can further help to modify existing code in a manner consistent with the original structure and understand the overall impact of such modifications. Any changes, especially those related to parameter usage within the body of a function, suggest the maintainer should consider the possibility of other "similar" functions being affected. This supports fast code modification risk assessment, before even performing regression tests which in practise are time consuming and often neglected. Maintainers should even be enabled to replace code sections of code without affecting functionality.

DMCC can also be used for perfective maintenance, when improving system cohesion and coherence by increasing modularity. This happens in two ways. Firstly, functions can be relocated within modules where they "naturally" belong. Secondly, processing within functions could be adjusted to better reflect the functionality that is supposed to be encapsulated within.

5. A method for program level comprehension

Concept assignment is a process aimed at assisting the maintainer in program comprehension by indicating where operations (e.g. Read) or entities (e.g. File) exist within the code. It involves identifying the location, scope, and instance of concepts within code. The type of concept assignment we are concerned with in this paper is termed *plausible-reasoning* owing to its use of multiple information sources (including informal clues such as comments) to assess the likelihood of the occurrence of a concept in the code. This approach differs from the common alternative of deriving the concepts from the semantics of the programming language (see section 3). The advantage of plausible-reasoning systems is their scalability over any size of program.

The Hypothesis-Based Concept Assignment (HB-CA) method [10], [11], [12] is a plausible-reasoning technique for identifying abstractions and concepts in COBOL code. Concepts are proposed by a maintainer and stored in a library as simple text strings. They are classified as either actions (i.e. they do something) or objects (they are something on which actions take place). Each concept has one or more indicators (also text strings) that, when found in code, may indicate the presence of the particular concept. Indicators are assigned to different classes: identifier (variable / procedure names), keyword, and comment (single words only, no phrases). Concepts can be joined by specialisation (one object to another) or in composition (one action with one object).

HB-CA is a three stage method comprising Hypothesis Generation, Segmentation, and Concept Binding. The library is used by the Hypothesis Generation stage to analyse the code and produce hypotheses for every concept whose indicators are found. The resulting hypothesis list is passed to the Segmentation stage which attempts to group hypotheses into coherent segments, each focussed around single concept. It uses the subroutine boundaries present in the original source code. Where the code has no subroutines or they are very large, a neural network is used to learn the conceptual structure of the hypotheses being considered and smaller segments defined based on this analysis. The segments are passed to the final stage: Concept Binding. This uses the weight of evidence for a concept (in terms of number of hypotheses) to determine which concept is dominant and thus present in the segment. If several concepts have the same level of evidence, a number of disambiguation rules are applied to pick a winner. The output is shown by colouring portions of the source code to match a coloured concept name displayed next to the code.

6. Satisfying the needs of software maintainers

As explained in section 2, despite existing methods and tools for system level and program comprehension, practitioners in the industry impose a set of requirements yet to be satisfied. Section 4 and 5 respectively introduced two methods, namely DMCC and HB-CA, facilitating these types of comprehension. We present here the way these methods individually address most of the above requirements. Furthermore, we discuss how coupling of the methods can satisfy the remaining requirements.

DMCC is an approach which successfully addresses the first two requirements set by the industry. It produces a high level overview of a system, where modules are

grouped together according to their similarity and their interrelationships are highlighted. It also provides the means to visualise and record a representation of a system, resembling a mental model which can be used to confirm perceptions, communicate these models and cross-reference them across a team. DMCC also provides maintainers with the required multi-layered subsystem abstraction which captures module interrelationships and can indicate the possible impacts of modifications.

HB-CA successfully addresses requirements 2, 3, 4, and 5. The need to share mental models is facilitated to some extent by the use and extension of the knowledge base by several maintainers. HB-CA provides a particularly good method for identifying the starting point for maintenance by providing the maintainer with a program representation in conceptual terms that they have nominated. The starting point can be expressed in terms closer to the problem. The shared knowledge base enables the recording of knowledge highlighted in requirement 4. Although the knowledge base structure is not elaborate, it does provide a mechanism by which maintainers can store parts of their system and program understanding for others to use. One of the main sources of knowledge for the HB-CA analysis is inline comments, used to determine which concepts are implemented in a particular section of code. It can be seen as a knowledge capturing method as desired in requirement 5.

The result of coupling DMCC and HB-CA addresses the rest of the requirements set by industrial practitioners, i.e. switching between System Level Comprehension and Program Comprehension (requirement 6) and accelerating and improving the quality of partial comprehension (requirement 7). The way these further requirements are met will be explained in the following section.

7. Combined method for better support

This section describes ways in which DMCC and HB-CA could be combined to improve the support offered to software maintainers.

DMCC gives an overview of the interrelations among low-level modules (functions) found in program files. Therefore:

- It can be used to assess modularity.
- It may be used for code ripple analysis and risk/impact analysis.
- It could be used prior to remodularization.

HB-CA gives an overview of the concepts found in a particular program file by mapping concepts (terms) to their implementation in code. Therefore:

- It can be used for business rule/code ripple analysis and risk/impact analysis.
- It can be used for module selection prior to change.
- It can be used to help with code reuse.
- It's useful in software module comprehension

There are several ways in which DMCC could be coupled with HB-CA to improve the completeness of comprehension support:

a. DMCC could assist in CA knowledge base generation. DMCC could be used to locate indicators (perhaps within the data sections of programs) and possibly concept-concept relationships. Concepts produced by DMCC are of "higher order" than the ones usually stored in the knowledge base. For example, instead of having a *read* concept, DMCC can introduce a *sort* concept which in fact consists of concepts of "lower order" such as *read, write* etc. This hierarchical approach extends the scope and enriches the usefulness of CA.

b. Segmentation could be based on DMCC "clusters" rather than regions of code formed between primary segmentation points or as an alternative to using neural network processing to find conceptual coherence. HB-CA initially segments code at section boundaries and then by use of Self-Organising Maps (SOMs) to reflect the conceptual structure of the program as expressed in terms of the knowledge base content. DMCC suggests further groupings of routines or paragraphs, which are more likely to contain "higher order" concepts and relationships.

c. Enhanced code ripple analysis and module selection. As both DMCC and CA may be used for code ripple analysis and risk/impact analysis results can be cross-validated when "overlapping" or combined when addressing different issues.

d. Cross-validation of DMCC and CA findings. This may happen if, instead of coupling the processes of the two methods, we only allow their results to be coupled. In other words, as DMCC produces high-level results and HB-CA produces low-level ones, there is a valid expectation that these can complement each other. This can be achieved by highlighting different aspects of a system or by providing two different angles for viewing a single aspect, lying in the boundaries of the scope of each method.

8. Conclusions and future work

System and program level comprehension is crucial for industrial scale software maintenance. A set of relevant requirements identified during a survey is only partly met by existing methods and tools. In this paper we have presented two methods that meet most of these needs individually. We have also proposed several ways in which they may be combined to greater effect and to provide more substantial support. This combination potentially addresses all the requirements.

There are a number of directions for further work in this area:

1) *Empirical validation of the combined approach.* It would be useful to expose the combined method to

maintainers in the real world to determine whether it can actually meet the needs identified in the early part of this paper.

2) *Closer integration between the methods.* The current style of coupling between the methods is loose and maintainers would benefit from a closer fit between them, as it would give them the ability to switch quickly between system views.
3) *Framework Development*. Many aspects of data mining are adopted in program comprehension tools and we plan to develop a framework to characterise and classify such tools by the data mining methods they adopt for data extraction and processing.

Acknowledgements

We gratefully acknowledge the support of EPSRC, the Leverhulme Trust, and CSC for various aspects of this work.

References

[1] F. Balmas, H. Wertz and J. Singer, "Understanding Program Understanding", Proc. 8th Int'l Workshop Program Comprehension (IWPC 00), IEEE Comp. Soc. Press, 2000, pp. 256.

[2] T.J. Biggerstaff, "Design Recovery for Maintenance and Reuse", IEEE Computer, Vol. 22, No. 7, July 1989, pp. 36-49.

[3] T.J. Biggerstaff, B. Mitbander, D. Webster, "The Concept Assignment Problem in Program Understanding", Proceedings of the Fifteenth International Conference on Software Engineering, Baltimore, Maryland, May 17-21, 1993, IEEE Computer Society Press, 1993, pp. 482-498.

[4] T.J. Biggerstaff, B.G. Mitbander, D.E. Webster, "Program Understanding and the Concept Assignment Problem", Communications of the ACM, Vol. 37, No. 5, May 1994, pp. 72-82.

[5] E. Burd, M. Munro, "Evaluating the Use of Dominance Trees for C and COBOL", Proceedings of the International Conference on Software Maintenance, Oxford, England, August 30-September 3, 1999, IEEE Computer Society Press, 1999, ISBN 0769500161, pp. 401-410.

[6] G. Canfora, A. Cimitile, A. De Lucia, G.A. Di Lucca, "Decomposing legacy systems into objects: an eclectic approach", Information and Software Technology, Vol. 43, 2001, pp 401-412.

[7] D.N. Chin, A. Quilici, "DECODE: A Cooperative Program Understanding Environment", Journal of Software Maintenance, Vol. 8, No. 1, 1996, pp. 3-34.

[8] P. Devanbu, R.J. Brachman, P.G. Selfridge, B.W. Ballard, "LaSSIE: A Knowledge-Based Software Information System", Communications of the ACM, Vol. 34, No. 5, May 1991, pp. 35-49.

[9] U. Fayyad, G. Piatetsky-Shapiro and P. Smyth, "From Data Mining to Knowledge Discovery: an Overview", Advances in Knowledge Discovery and Data Mining, AAAI Press, 1996, pp. 1-34.

[10] N.E. Gold, "Hypothesis-Based Concept Assignment to Support Software Maintenance", PhD. Thesis, Department of Computer Science, University of Durham, 2000.

[11] N.E. Gold and K.H. Bennett, "A Flexible Method for Segmentation in Concept Assignment", Proc. Int'l Workshop on Program Comprehension (IWPC 01), IEEE Comp. Soc. Press, 2001.

[12] N.E. Gold, "Hypothesis-Based Concept Assignment to Support Software Maintenance", Proc. Int'l Conference on Software Maintenance (ICSM 01), IEEE Comp. Soc. Press, 2001.

[13] M.T. Harandi, J.Q. Ning, "Knowledge-Based Program Analysis", IEEE Software, Vol. 7, No. 1, January 1990, pp. 74-81.

[14] J. Hartman, "Automatic Control Understanding for Natural Programs", Ph.D. Thesis, University of Texas at Austin, May 1991.

[15] A. K. Jain and R. C. Dubes, Algorithms for Clustering Data, Prentice-Hall, 1988.

[16] W.L. Johnson, Intention-Based Diagnosis of Novice Programming Errors, Morgan Kaufmann Publishers Ltd, 1986, ISBN 0273087681.

[17] V. Karakostas, "Intelligent Search and Acquisition of Business Knowledge from Programs", Software Maintenance: Research and Practice, Vol. 4, 1992, pp. 1-17.

[18] W. Kozaczynski, J.Q. Ning, "Automated Program Understanding By Concept Recognition", Automated Software Engineering, Vol. 1, No. 1, March 1994, pp. 61-78.

[19] T.M. Pigoski, *Practical Software Maintenance: Best Practices for Managing your Software Investment*, Wiley Computer Publishing, 1996.

[20] C. Rich, R.C. Waters, The Programmer's Apprentice, ACM Press (Frontier Series), 1990, ISBN 0201524252.

[21] M. Siff, T. Reps, "Identifying Modules via Concept Analysis", IEEE Transactions on Software Engineering, Vol. 25, No. 6, November/December 1999.

[22] I. Sommerville, *Software Engineering*, 6th edition, Harlow, Addison-Wesley, 2001.

[23] F. Tip, "A Survey of Program Slicing Techniques", Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam, 1994.

[24] C. Tjortjis, "Using Data Mining for Program Comprehension", PhD. Thesis, Department of Computation, UMIST, to appear 2002.

[25] C. Tjortjis and P.J. Layzell, "Using Data Mining to Assess Software Reliability", Suppl. Proc. IEEE 12th Int'l Symposium Software Reliability Engineering (ISSRE2001), IEEE Comp. Soc. Press, 2001, pp. 221-223.

[26] C. Tjortjis and P.J. Layzell, "Expert Maintainers' Strategies and Needs when Understanding Software: A Qualitative Empirical Study", Proc. IEEE 8th Asia-Pacific Software Engineering Conf. (APSEC 2001), IEEE Comp. Soc. Press, 2001, pp. 281-287.

[27] L.M. Wills, "Automated Program Recognition by Graph Parsing", PhD Thesis, AI Lab, Massachusetts Institute of Technology, July 1992.

[28] S.G. Woods, A.E. Quilici, Q. Yang, Constraint-Based Design Recovery for Software Reengineering: Theory and Experiments, Kluwer Academic Publishers, 1998, ISBN 0792380673.